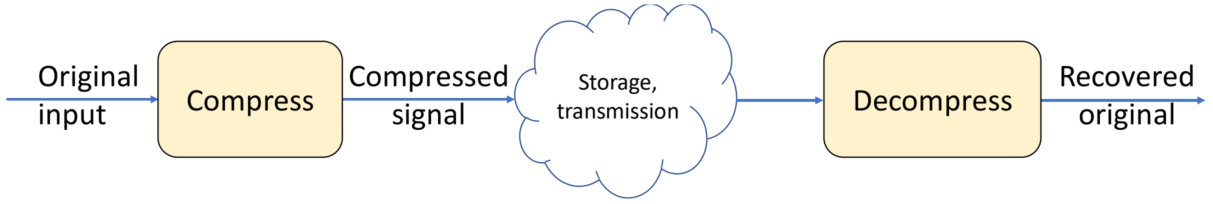
CS112 - Fall 2022

Project02 – Text Compression via Huffman Coding

INTRODUCTION

For **Project02** you will develop programs that compress and decompress text files. Data compression is a critical technology—it enables significant cost and resource savings for many data storage and transmission applications:

- Storage of audio and video on DVDs, Blu-rays, etc
- Audio and video streaming, e.g. by Netflix, Spotify, etc
- Audio compression for cellular telephony, digital radio, etc
- Computer data storage and transmission



There are two general categories of compression technologies:

- Lossless – the decompressed results are identical to the original data before compression
- Lossy – the decompressed results are very similar to but are not identical to the original

Why would anyone ever use lossy compression? Lossless techniques can achieve "compression ratios" (size of original data / size of compressed data) of 2:1 or maybe 3:1. Today it is common for High Definition video to be compressed with lossy compression almost 1000:1.

In this project, we will learn about one basic lossless compression technology called "Huffman encoding". We will use that technology to develop programs for lossless compression and decompression of text files.

Morse Coding

Lossless compression methods all rely on the fact that some **input symbols** (text characters in our case) occur more often than others. For example in English text, the letter 'e' occurs much more often than the letter 'q'. If we use fewer bits to transmit a compressed 'e' than we use for a compressed 'q', then since we send the letter 'e' so often, we should be able to compress text compared to the original. Of course, the

compressed symbol for every letter of the alphabet should be chosen to reflect the letters' probabilities: long compressed symbols for less common letters, and short compressed symbols for frequently occurring letters.[1]

Probably the most famous compression technique for English text is Morse Code, developed by inventors Alfred Vail and Samuel F. B. Morse for use with the telegraph.[2] Morse Code encodes every Latin-alphabet letter and every numeral from 0 to 9 with a series of brief "dots" and longer "dashes". These can be transmitted electrically as two different voltages, as sounds, light flashes, etc. In modern Morse Code, the compressed symbol for the letter 'e' is a single dot "·" and the compressed symbol for 'q' is "- - · -".

In Morse Code, the compressed symbol for the letter 's' is "· · ·". Do you see the problem with this?

[1] Ideally, the length of the coded symbol for each letter should be $-\log_2$(Probability of the letter), so if the probability is 0.5, the length is 1. If probability = 0.25, length = 2, etc.
[2] https://en.wikipedia.org/wiki/Morse_code, downloaded July 20, 2022

A Problem with Morse Code

If the compressed symbol for ‘s’ is “· · ·” and the compressed symbol for ‘e’ is “·”, then how can a receiver distinguish an ‘s’ from the pattern ‘eee’ ? In fact, Morse Code uses <u>three</u> values in its **compressed alphabet** (each **compressed symbol** is a combination of one or more **characters** from the **compressed alphabet** in a specific order):

- Dot
- Dash
- silence

The duration of silence between the dots and dashes is significant: the three dots in the symbol for ‘s’ occur closer together than the three dots in the symbol for ‘eee’. This is inefficient and is certainly difficult for computers, which work natively in binary. But it works well for human Morse code experts, who like a space between letters, a longer space between words, and a longer space between sentences.

Huffman Coding

Huffman Coding is similar in concept to Morse Code, but Huffman Coding truly uses only two characters in its compressed alphabet, which are usually denoted as ‘0’ and ‘1’. The key to the design of Huffman Coding is that the compressed symbol for each input symbol is <u>not a prefix for</u> the compressed symbol of any other input symbol. Once we have seen some pattern of ‘0’s and ‘1’s that matches the compressed symbol for a particular input, we do not have to guess whether the pattern indicates our input or is the beginning of the symbol for some other input. Here is an example of a Huffman Code for five possible inputs that shows this property:

Input	**Compressed Symbol**
α	00
β	01
γ	100
δ	101
ε	11

Let’s verify this “no-prefix” property…

- If the first value you receive is a ‘0’ then the next received value tells you whether the input was ‘α’ or ‘β’.
- If you first receive a ‘1’ then wait for the next received value
    - If it is a ‘1’ then the input was an ‘ε’

    - If it is a ‘0’ then the third received value determines whether the input was ‘γ’ or ‘δ’
- The next received value <u>must</u> be the start of the compressed symbol for the <u>following</u> input character

Construction of Huffman Codes

We will construct a Huffman code that processes one input text character at a time. As part of that construction, we will estimate the probabilities of occurrence of different input text characters by counting how many times each character value occurs in a “training file” of text. Remember our earlier character counting program? We can reuse parts of that here.

Huffman Code Algorithm

There is a clever algorithm for the construction of Huffman codes that relies on trees and lists:

- First, for each allowed character in our input, generate a measure of the probability of occurrence for each character. We do that by counting how many times each character occurs in a "training file" of text data. Remember our earlier character-counting program? We can reuse much of that here.
- Next, form a list of “Nodes”. Each Node contains
    - A unique input character (i.e. we have a Node for each input character)
    - A ‘count’ for how often the character is expected. We got these from the first step.
    - A ‘coded symbol’ – the variable length compressed symbol used in Huffman coding. Below I will explain how the symbol is found for each character. This can be stored as a String.
- Next, sort the list, in order of increasing count. You can do the sorting yourself or you can use Java's built-in sorting method.

Ok, on to the next steps!

- As long as the list has more than 1 element, perform the following steps repeatedly:
    - Remove the two smallest-count Nodes from the list. Create a new Node that points to both of the removed Nodes. (So the Node class needs references to two other Nodes—we are going to build a tree!) The new Node does not have a real character value but does have a count, which equals the sum of the counts of the two Nodes that the new Node points to.
    - Insert this new Node into your original list. Now the list has one fewer elements, but some of the elements are Nodes that point to other Nodes (and those other Nodes no longer belong to the list)
    - Again sort the list in order of increasing count

Great. After the above loop, your list has only one Node. And <u>that Node is the root of a binary tree</u>. The root Node has two children, and its children probably have children also. But the "leaves" of the tree are all Nodes that have actual input character values (and do not have any children).

- All Nodes in the tree have either 2 children or 0 children. Give this some thought to confirm!

Next we assign coded symbols to all leaf nodes:

- First, assign the root node the symbol `""`
- Now use recursion!
    - For every node in the tree, assign its "left-side" child (if it has a child) the symbol whose value is the parent node's symbol + `"0"`
    - For every node in the tree, assign its "right-side" child (if it has one) the symbol whose value is the parent node's symbol + `"1"`

It is worth sketching this out on a sheet of paper with a small tree, so you understand what is happening.

Now every "leaf" node has a symbol that is unique, and further, is not a prefix for any other value's symbol! Wow—you did it!

With this Huffman tree, decoding is pretty straightforward:

- Have a Node pointer point to the top of the Huffman tree, and read a "0" or "1" from the compressed input
- If the value read is "0", then change your pointer so it points to the left-side child of the pointed-to Node
- If the bit read is "1", then point to the right-side child of the pointed-to Node
- If the resulting node is a "leaf node", output the corresponding input character, and then start decoding the next character
- If the resulting node is still not a "leaf node", then read another value from the compressed input and traverse further down the tree

Encoding is also pretty easy. For convenience, rather than a tree, we will want a Dictionary i.e. HashMap of all input character values and their symbols. Use recursion to traverse the Huffman tree and generate that Dictionary. Then, to encode:

- Read an uncompressed input character value. Look up the value's compressed symbol in the dictionary and output the compressed symbol as a sequence of text "1"s and "0"s
- Move on to the next input character

Your Programs

This project consists of three programs:

- **Generate** - Generate a Huffman table, given training data
- **Encode** - Compress text files, given the Huffman table from "Generate"
- **Decode** – Decompress compressed files, hopefully yielding the original text files

The **Generate** program takes a single command-line input: the name of the text "training file" that is used to generate counts of how often each allowed input character value occurs. **Generate** shall generate a Huffman code. The output of Generate is a text file called **codebook** which shall be saved in the same directory as holds the **Generate.java** program.

The format for the **codebook** file essentially is the dictionary used by "Encode". The file shall consist of multiple lines, one line for each allowed input character value. Each line shall have the following format, all fields expressed as ASCII (equivalent to UTF-16) characters:

<the character's UTF value as a decimal integer><colon><compressed symbol for the character as text "0"s and "1"s><newline>

For example:

61:1100111

The **Encode** program shall take two command-line arguments: the name of an uncompressed text file and the name of an output file that holds the Huffman-coded version of the input. **Encode** reads the Huffman table from the **codebook** file, of course. The output file shall hold a pattern of text "0"s and "1"s that represents the input in compressed form. The **Encode** program shall send one Huffman-coded *EOT* character (see below) after processing all input characters, and the **Decode** program shall stop processing its input once it sees the *EOT*. In **Generate**, be sure to include the *EOT* character with a count of 1.

The **Decode** program also takes two command-line arguments: the name of a Huffman-compressed input file, and the name of a text output file. **Decode** reads its Huffman table from the **codebook** file. **Decode** performs Huffman decoding of its compressed input and outputs the decompressed results. It should read but not output the *EOT* character; after reading the *EOT*, **Decode** can close its files and exit.

There is one challenge you must overcome when you write the **Decode** program: the **codebook** file contains our Huffman code as a Dictionary, not as a Tree. You must reconstruct the tree from the dictionary. Please think about how to do this to see if you can figure out how on your own. I'll give hints if asked.

Project Details

For this project we will support only the following set of uncompressed text character values:

- `'\u0004'`: the *EOT* "End Of Transmission" character. The compressor shall send this after processing all its input, to signal to the decompressor that the input has ended

- `'\u0007'` through `'\u00fe'` inclusive. This is a small subset of UTF-16 but is larger than the old ASCII character set.

If any inputs have values other than these, **Encode** shall skip over them on the input, and **Generate** shall ignore them.

Grading

The main goal is for your decoder output to match your encoder input exactly. You also should be able to achieve 20% - 40% compression on English text files. Don't worry about trying to improve the algorithm! If you implement the algorithm, you will get compression.

If you look at the size of your input file and your compressed file, why is the compressed file so big? It stores '0' and '1' values as text, using 8 bits (probably) for each character. In an actual data compression or transmission application, we would handle the '0's and '1's as individual bits, and your compressed file would be 1/8 as large. (Extra credit??? Just kidding—no.)

Put your three programs into a subdirectory called **Project02** inside your **MyWork** directory, and remember to push your **Project02** to GitHub before the deadline.

As with Project01, please prepare a 1 page writeup on how you tested your programs, and push that document to GitHub

- The project is due before 11:59pm on Friday December 2nd.

Rubric

Milestone	Points	Comments
Programs are in correct location, pushed to GitHub before deadline, and compile successfully	0	
Generate generates a Huffman file with the correct format	5	
Encode compresses input textfiles (almost) as much as my reference implementation. And **Decode** produces output that matches the original input exactly	5 test cases, 5 points each	
Software quality	20	Judged subjectively by graders
1-page informal writeup of what and how you tested your programs.	20	Judged subjectively by graders

Conclusion

This project ties together many of the techniques you have practiced during the course:

- File processing
- Text manipulation
- Advanced data structures
- Recursion
- Programming for Robustness
- Excellent test planning and execution

You should be able to reuse several classes and methods developed earlier in the Labs to help you with this project.

Congratulations on a job well done!